

## Fast Hyperbolic functions for Arbitrary Precision numbers.

By Henrik Vestermark (hve@hvks.com)

### Abstract:

This paper is a follow-up to a previous paper that describes the math behind arbitrary precision numbers, see [7]. First, the original paper was written in 2013, and quite a few things have happened since then. Secondly, I came across other interesting methods for calculating the hyperbolic function. The paper describes in more detail how to do the hyperbolic functions  $\sinh(x)$ ,  $\cosh(x)$ ,  $\tanh(x)$  and the inverse  $\operatorname{arsinh}(x)$ ,  $\operatorname{arcosh}(x)$ , and  $\operatorname{artanh}(x)$  calculation with arbitrary precision. Furthermore, we outline some traditional methods and introduce an improved version that makes the calculation 5-10 times faster than the original method used in the author's arbitrary precision math packages.

### Introduction:

When implementing arbitrary precision math packages, you would use the standard Taylor series for calculating  $\sinh(x)$  and  $\cosh(x)$  for arbitrary precisions. In contrast,  $\tanh(x)$ ,  $\operatorname{arsinh}(x)$ ,  $\operatorname{arcosh}(x)$ , and  $\operatorname{artanh}(x)$  can be derived from  $\exp(x)$  or  $\log(x)$ . The Taylor series for hyperbolic functions is not particularly fast in its raw form. However, you can apply techniques that significantly improve the method's performance. We will discuss the various methods for calculating hyperbolic functions and elaborate on techniques like clever argument reductions and coefficient scaling to enhance the method's performance.

We will show the actual C++ source for the computation using the author's arbitrary precision Math library (see [1]).

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at [www.hvks.com/Numerical/papers.html](http://www.hvks.com/Numerical/papers.html) and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
3. Fast Square Root & Inverse calculation for arbitrary precision math. [HVE Fast Square Root & inverse calculation for arbitrary precision](#)
4. Fast Exponential calculation for arbitrary precision math. [HVE Fast Exp\(\) calculation for arbitrary precision](#)
5. Fast logarithm calculation for arbitrary precision math. [HVE Fast Log\(\) calculation for arbitrary precision](#)
6. Practical implementation of Spigot Algorithms for Transcendental Constants. [Practical implementation of Spigot Algorithms for transcendental constants](#)
7. Practical implementation of  $\pi$  algorithms. [HVE Practical implementation of PI Algorithms](#)

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

8. Fast Trigonometric function for arbitrary precision. [HVE Fast Trigonometric calculation for arbitrary precision](#)
9. Fast Hyperbolic functions for arbitrary precision. [HVE Fast Hyperbolic calculation for arbitrary precision](#)
10. Fast conversion from arbitrary precision number to a string. [HVE Fast conversion from arbitrary precision to string](#)
11. Fast conversion from a decimal string to an arbitrary precision number. [HVE Fast conversion from string to arbitrary precision](#)

## Change log

25-February 2023. Cleaning up the grammar and correcting minor inaccuracies.

9-June 2024. Added two other methods to compute  $\sinh(x)$  and  $\cosh(x)$

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## Contents

Abstract: .....	1
Introduction: .....	1
Change log .....	2
The Arbitrary precision library .....	5
Internal format for float_precision variables .....	6
Normalized numbers .....	6
Hyperbolic functions .....	8
Sinh(x) using Exp(x) .....	8
Sinh(x) using the Taylor series .....	9
Example sinh(1): .....	9
The issue with arbitrary precision .....	9
Argument Reduction .....	10
Example – Two-argument reduction: .....	11
Example – Eight-argument reductions: .....	11
Finding a reasonable argument reduction factor .....	11
Guard Digits .....	12
Further improvements of the method? .....	13
No coefficient scaling .....	13
Partial coefficient scaling .....	13
Full coefficient scaling .....	13
Partial coefficient scaling .....	14
Source for sinh(x) using coefficient scaling .....	14
Full coefficient scaling .....	16
sinh(x) Performance .....	19
Recommendation for calculating sinh (x) .....	21
Cosh(x) using Exp(x) .....	21
Example of Cosh(x) using exp(x) .....	22
Cosh(x) using Taylor series: .....	22
Example cosh(1): .....	22
Argument Reduction .....	23
Example – Two-argument reduction: .....	23
Example – Eight-argument reductions: .....	23
Cosh(x) using double angle reduction .....	24
Source for cosh(x) with argument reduction and coefficient scaling .....	24
Coshx(x) with full coefficients scaling .....	27
Cosh(x) Performance .....	30
Recommendation for calculating cosh(x) .....	32
Tanh(x): .....	32
Source for tanh(x) .....	33
Recommendation for calculating tanh(x) .....	33
Arcsinh(x): .....	33
Arcsinh(x) direct method: .....	34
Source for asinh(x) .....	34
Arcsinh(x) using the Taylor series .....	34

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

Example: Arcsinh(0.1).....	34
Example: Arcsinh(0.7).....	35
Recommendation for calculating Arcsinh(x).....	36
Arccosh(x): .....	36
Arccosh(x) direct method: .....	36
Source for Arccosh(x).....	36
Arccosh(x) using the Taylor series .....	37
Example: Arccosh(5) .....	37
Example: Arccosh(2) .....	37
Recommendation for calculating Arccosh(x) .....	38
Arctanh(x): .....	38
Arctanh(x) direct method.....	38
Source for Arctanh(x) .....	39
Arctanh(x) using the Taylor series.....	39
Example Arctanh(0.1).....	39
Example Arctanh(0.5).....	40
Recommendation for calculating Arctanh(x).....	41
Overall Recommendation for calculating Hyperbolic functions .....	41
Reference .....	42

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## The Arbitrary precision library

If you are already familiar with the arbitrary precision library, you can skip this section.

To understand the C++ code and text, we must highlight a few features of the arbitrary precision library, such as the class name *float\_precision*. Instead of declaring a variable with a float or double, you replace the type name with *float\_precision*. E.g.

```
float_precision f; // Declare an arbitrary precision float with 20 decimal digits precision
```

You can add a few parameters to the declaration. The first is the optional initial value, and the second optional parameter is the floating-point precision. The native float type has a fixed size of 4 bytes and 8 bytes for *double*. However, since this precision can be arbitrary, we can declare the desired precision as the number of **decimal digits** we want to use when dealing with the variable. E.g.

```
float_precision fp(4.5); // Initialize it to 4.5 with default 20 digits precision
float_precision fp(6.5,10000); // Initialize it to 6.5 with a precision of 10,000 digits
```

The precision of a variable can be dynamic and change throughout the code, which is very handy to manipulate the variable. To change or set the precision, you can call the method `.precision()`, for example,

```
f.precision(100000); // Change the precision to 100,000 digits
f.precision(fp.precision()-10); // Lower the precision with 10 digits
f.precision(fp.precision()+20); // Increase precision with 20 digits
```

There is another method to manipulate the exponent of the variables. The method is called `.exponent()` and returns or sets the exponent as a power of two exponents (same as for our regular build-in types *float* and *double*), E.g.

```
f.exponent(); // Return the exponent as 2e
f.exponent(0) // Remove the exponent
f.exponent(16) // Set the exponent to 216
```

There is a second way to manipulate the exponent, and that is through the class method `.adjustExponent()`. This method adds the parameter to the internal variable that holds the exponent of the *float\_precision* variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number with 2.
f.adjustExponent(-1); // Subtract one from the exponent, the same as dividing the number with 2.
```

This allows very fast multiplication or division with a number that is any power of two.

The method `.iszero()` returns true if the *float\_precision* number is zero; otherwise false.

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

There is an additional method(), but I will refer to the reference for the user manual and the arbitrary precision math package for details.

All the regular operators and library calls that work with the built-in float or double will also work with the float\_precision type using the same name and calling parameters.

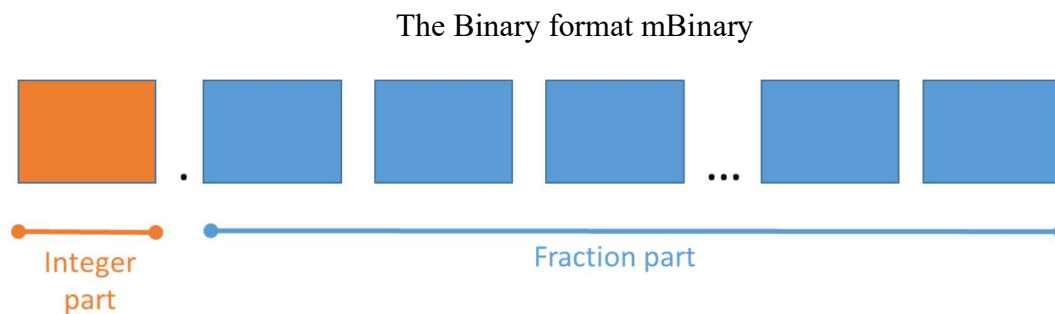
## Internal format for float\_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mBinary;
```

*uintmax\_t* is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integers to store our floating-point precision number.

The method .size() returns the number of internal vector entries needed to hold the number.



- The binary format consist of an unlimited number of 64bit unsigned integer blocks.
- One block in front of the period sign ‘.’ (the integer part of the number)
- Zero or more blocks of fractions after the ‘.’ (the fraction sign of the number)
- The binary number is stored in a STL vector class and defined
  - `vector<uintmax_t> mBinary;`
- There is always one entry in the mBinary vector.
- Size of vector is always  $\geq 1$
- A Number is always stored normalized. E.g. the integer part is 1 or zero
- The sign, exponent, precision, rounding mode is stored in separate class fields.

Other internal class variables, such as the sign, exponent, precision, and rounding mode, are unimportant for understanding the code segments.

## Normalized numbers

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

A `float_precision` variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details see [1].

## Hyperbolic functions

Usually, you use a Taylor series to calculate the Hyperbolic functions for  $\sinh(x)$  and  $\cosh(x)$  and some simple hyperbolic identity to calculate  $\tanh(x)$ ,  $\operatorname{arcsinh}(x)$ ,  $\operatorname{arccosh}(x)$ , and  $\operatorname{arctanh}(x)$ . This chapter will examine:

- 1)  $\operatorname{Sinh}(x)$  using  $\exp(x)$
- 2)  $\operatorname{Sinh}(x)$  using the Taylor series, argument reduction, and coefficient scaling.
- 3)  $\operatorname{Cosh}(x)$  using the Taylor series, argument reduction, and coefficient scaling.
- 4)  $\operatorname{Tanh}(x)$  using a simple identity.
- 5)  $\operatorname{Arcsinh}(x)$  using a simple identity.
- 6)  $\operatorname{Arccosh}(x)$  using a simple identity.
- 7)  $\operatorname{Arctanh}(x)$  using a simple identity.

The standard Taylor series expansion method is the most common for arbitrary precision libraries.

## $\operatorname{Sinh}(x)$ using $\operatorname{Exp}(x)$

It is tempting to use the definition of  $\operatorname{Sinh}(x)$ :

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right) \quad (1)$$

We only need to calculate  $e^x$  once. In particular, if you have a fast implementation of  $e^x$ , you can use the above to calculate  $\sinh(x)$  and save some code. However, recall ref [4] where the recommended method for calculating  $e^x$  is to use the sine hyperbolic function:

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (2)$$

If you have implemented the above method for  $\exp(x)$ , then you will experience a slightly slower performance when using  $\exp(x)$  to calculate  $\sinh(x)$ . Usually,  $\sinh(x)$  is faster to compute than  $\exp(x)$ .

Example of  $\sinh(x)$  using  $\exp(x)$

```
float_precision sinhExp(const float_precision& x)
{
    float_precision v;
    const float_precision c1(1);

    v.precision(x.precision());
    v = exp(x);
    v -= c1 / v;
    v.adjustExponent(-1);    // v *= 0.5;

    // Round to same precision as argument and rounding mode
    v.mode(x.mode());
}
```

```
v.precision(x.precision());  
return v;  
}
```

## Sinh(x) using the Taylor series

We have already seen that using the sinh(x) Taylor series for calculating  $e^x$  is faster than the  $e^x$  using the Taylor series. See ref [4]. We will repeat the finding from ref [4] below.

Sinh(x) is found with the Taylor series:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (3)$$

Example sinh(1):

We are calculating sinh(1) using no argument reduction. We need seven Taylor terms to get the result using the Taylor series.

Sinh(x)		Original	X Reduced	
x=		1	1	
Taylor reductions=		0		
Terms	Term value	Term Sum	Sinh(x)	Error
1	1.00E+00	1.00000000000	1.0000000000	1.75E-01
2	1.67E-01	1.16666666667	1.1666666667	8.53E-03
3	8.33E-03	1.17500000000	1.1750000000	2.01E-04
4	1.98E-04	1.17519841270	1.1751984127	2.78E-06
5	2.76E-06	1.17520116843	1.1752011684	2.52E-08
6	2.51E-08	1.17520119348	1.1752011935	1.61E-10
7	1.61E-10	1.17520119364	1.1752011936	7.67E-13

## The issue with arbitrary precision

The number of Taylor terms to reach a result does not seem too bad at first glance. In the previous examples, we only used approx. Seven Taylor terms. However, when dealing with higher precisions, e.g., 1,000, 10,000, or even 100,000 digits, we suddenly have to perform many more Taylor terms to find our result. In Yacas' book of algorithms [5], they found a bound for the number of Taylor terms,  $n$ , needed for the  $\sin(x)$  as a function of the number of precisions in digits  $P$  and the magnitude,  $M$ , of the argument  $x=10^M$ . You can use the same rationale as they used for  $\sin(x)$  to get a bound for the number of Taylor terms for  $\sinh(x)$ :

$$2(n + 1) \approx \frac{(P - M) \cdot \ln(10)}{\ln(P - M) - 1 - M \cdot \ln(10)} \Rightarrow$$

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

$$n \approx \frac{1}{2} \frac{(P-M) \cdot \ln(10)}{\ln(P-M) - 1 - M \cdot \ln(10)} - 1 \quad (4)$$

The number of Taylor terms needed for sinh(x) as a function of precision and argument magnitude.

Digits	10 <sup>1</sup>	10 <sup>2</sup>	10 <sup>3</sup>	10 <sup>4</sup>	10 <sup>5</sup>	10 <sup>6</sup>	10 <sup>7</sup>	10 <sup>8</sup>
<b>x</b>								
<b>10<sup>1</sup></b>	(11)	88	319	1,948	14,022	109,512	898,358	7,615,327
<b>10<sup>0</sup></b>	8	31	194	1,402	10,951	89,835	761,532	6,608,768
<b>10<sup>-1</sup></b>	3	19	140	1,095	8,983	76,153	660,876	5,837,230
<b>10<sup>-2</sup></b>	2	14	109	898	7,615	66,087	583,723	5,227,006
<b>10<sup>-3</sup></b>	1	11	90	761	6,608	58,372	522,700	4,732,291
<b>10<sup>-4</sup></b>	1	9	76	661	5,837	52,270	473,229	4,323,125
<b>10<sup>-5</sup></b>	1	7	66	584	5,227	47,323	432,312	3,979,084
<b>10<sup>-6</sup></b>	1	6	58	522	4,732	43,231	397,908	3,685,765
<b>10<sup>-7</sup></b>	1	6	52	473	4,323	39,791	368,576	3,432,721
<b>10<sup>-8</sup></b>	1	5	47	432	3,979	36,857	343,272	3,212,190
<b>10<sup>-9</sup></b>	(1)	5	43	398	3,686	34,327	321,219	3,018,284

The table above is quite interesting. For, the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of 1 in magnitude down to arguments of 10<sup>-9</sup>. The factor is only around three for a precision of 100,000 digits; for 100M digits, it is around 2.2. The lesson is that argument reduction is more efficient for smaller than higher precision. However, overall argument reduction is beneficial at any precision. There is another approximation for *n* based on the actual value of x, not just the magnitude. It usually gives a little bit less of the needed Taylor terms. This formula can be pretty helpful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (5)$$

## Argument Reduction

Looking at the Taylor series for sinh(x), it is clear that we prefer to have our  $|x| < 1$  to ensure that the Taylor series converges more quickly. As we have seen before, we can use *argument reduction* to work with a smaller number to converge sinh(x) faster using fewer *Taylor terms* of the Taylor series.

We can use the trisection identity:  $\sinh(3x) = \sinh(x)(3 + 4\sinh^2(x))$  to reduce the argument with a factor of three, and then after the Taylor iterations, we restore and find the correct value for sinh(x) by applying this formula the same number of times we did when reducing the argument.

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## Example – Two-argument reduction:

Using the same example as before for  $\sinh(1)$  and using two argument reductions, you get the result after only four Taylor terms compared to seven with no argument reductions.

Sinh(x)		Original	X Reduced	
x=		1	0.111111111	
Taylor reductions=		2		
Terms	Term value	Term Sum	Sinh(x)	Error
1	1.11E-01	0.111111111111	1.1720460995	3.16E-03
2	2.29E-04	0.11133973480	1.1751992452	1.95E-06
3	1.41E-07	0.11133987592	1.1752011931	5.73E-10
4	4.15E-11	0.11133987596	1.1752011936	9.84E-14

## Example – Eight-argument reductions:

With eight times argument reduction, you get the result after two Taylor terms compared to four using two argument reductions.

Sinh(x)		Original	X Reduced	
x=		1	0.000152416	
Taylor reductions=		8		
Terms	Term value	Term Sum	Sinh(x)	Error
1	1.52E-04	0.00015241579	1.1752011877	5.97E-09
2	5.90E-13	0.00015241579	1.1752011936	0.00E+00

Unsurprisingly, using argument reduction significantly reduced the number of Taylor terms needed and will result in faster performance.

## *Finding a reasonable argument reduction factor.*

As the above table shows, a higher reduction factor greatly improved the performance. However, how many times of argument reduction is adequate? The argument reduction on the front end is a division per reduction. In the back end, you do this as many times as you did the reductions on the front end.  $\sinh(3x)=3\sinh(x)-4(\sinh^3(x))$  taking  $\sinh(x)$  out as a factor you get this:  $\sinh(3x)=\sinh(x)(3-4(\sinh^2(x)))$  or one subtraction and three multiplication. Using

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (6)$$

At a starting point of  $x=1$ , you get for  $P=100$  digits that the needed Taylor term is 24. Doing three reductions, you get  $x=1/3^3 = 0.037$ . We expect to use the above formula only to need 14 Taylor terms. Each Taylor term requires one addition/subtraction, 1 division, and multiplication, which yields a total saving of 10 subtractions, 10 divisions, and 10

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

multiplication. Compared to three reductions on the front end, there are three divisions, and on the back end, 3 subtractions and nine multiplications, a total saving of seven subtractions/additions, one multiplication, and seven divisions. Since division is slower than multiplication and addition/subtraction, we can give a rough saving equivalent of seven divisions. For higher precisions, the savings become larger.

We automatically calculate the reduction factor as follows:

$$k = 8 \left\lceil \frac{2}{3} \ln(2) * \ln(P) \right\rceil \quad (7)$$

For higher precisions, we made adjustments for the magnitude of  $x$ . We add the exponent to the reduction factor. If  $x$  is large, we do more argument reductions; if  $x$  is small, we reduce the number of reductions. This means that our reduction factor gets smaller if  $x$  is very small, preventing us from making unnecessary reductions. If  $x$  is very small, the reduction factor is negative, and we do not perform any argument reductions. For example,  $P=100$ , you get 24, and for  $P=10,000$ , you get 40. To compensate for inaccuracies when adding the front and back end calculation, we increase the precision by the reduction factor,  $k/4$ . The increased precision only generates a minor performance penalty compared to the extra saving in Taylor's terms of the overall calculation.

To calculate a reasonable reduction factor, we make it a function of the desired precision and the magnitude of the argument  $x$ . For example, argument reduction increased as a log function of the wanted precision, and argument reduction increased with a large magnitude of the number and decreased for a smaller magnitude of argument  $x$ .

We use the following code segment in our source code to calculate the needed argument reduction.

```
// automatically calculates the optimal reduction factor as a power of two
// and as a function of the wanted precision
k = 8 * (intmax_t)ceil(log(2)*log(precision));
// Now adjust for small or large arguments.
k += v.exponent() + 2;
k = std::max((intmax_t)0, k);
//Since we are using the trisection identity:
//     sinh(3x)=sinh(x)(3+4Sinh^2(x))
//We reduce the argument to 2/3*k times.
// Converting power of 2 to power of 3. (roughly)
k = (intmax_t)ceil(2.0*k / 3);
```

## Guard Digits

When summarizing a Taylor series as  $\sinh(x)$ , you need quite a lot of summarizing, which will produce round-off errors.

For our  $\sinh(x)$  function, we use a simple guard digits calculation that we add.

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

$2 + \lceil \log_{10}(\text{precision}) \rceil$  as extra guard digits for working precision.

## ***Further improvements of the method?***

Coefficient scaling can be considered an improvement over the standard method.

Generally speaking, you can have three options here.

- No coefficient scaling
- Partial coefficient scaling
- Full coefficient scaling

### No coefficient scaling

No coefficient scaling is just the regular use of the Taylor series, as presented above. Each Taylor term is computed, including the division, which performs poorly compared to the other operators, particularly in arbitrary precision libraries. Although the method is straightforward, it ensures that each term is as accurate as possible before contributing to the final sum. It can be computationally expensive due to repeated division operations, mainly when used in connection with arbitrary precision libraries.

### Partial coefficient scaling

You can group more Taylor terms before dividing. For example, you reduce the number of division operations by calculating five Taylor terms at a time and then dividing the sum of these terms by the appropriate factorial. Partial coefficient scaling will enhance performance since division is generally more computationally intensive than multiplication or addition. The trade-off here is a potential slight decrease in numerical precision, as errors could accumulate in the grouped terms before the division normalizes them.

### Full coefficient scaling

When doing a full coefficient scaling, you postpone the division until all Taylor terms have been computed. This method involves summing all scaled Taylor series terms first and dividing the final sum by the factorial. This method could be the fastest in reducing the number of division operations, but it may also lead to the most significant error. When terms with large magnitudes are summed without immediate normalization, floating point errors can accumulate, particularly for higher-order terms or larger values of  $x$ .

The discussion of partial and full coefficient scaling is as follows.

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## Partial coefficient scaling

The same technique for coefficient scaling (grouping of Taylor terms) can also be applied to the  $\sinh(x)$  here. Consider the Taylor series for sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (8)$$

The issue again clearly is the division for each term. Since division is many times slower than calculation and addition, you could group two or more Taylor terms (sometimes called coefficient scaling) and reduce the number of divisions. Consider the  $n^{\text{th}}$  and the  $n+1$  terms:

$$\dots \frac{x^n}{n!} + \frac{x^{n+2}}{(n+2)!} \dots$$

Moreover, group them:

$$\begin{aligned} \dots \frac{(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \dots &=> \\ \dots \frac{(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \dots & \end{aligned}$$

Then, you have replaced one division with two extra multiplications. The  $(n+1)(n+2)$  can be done using 64-bit integer arithmetic since you never get to do so many Taylor terms in real life that it will overflow. There is no need to stop at just grouping two terms. You can do that for three terms:

For grouping three Taylor terms, you get:

$$\begin{aligned} \dots \frac{(n+1)(n+2)(n+3)(n+4)x^n + (n+3)(n+4)x^{n+2} + x^{n+4}}{(n+4)!} \dots &=> \\ \dots \frac{(n+3)(n+4)((n+1)(n+2)x^n + x^{n+2}) + x^{n+4}}{(n+4)!} \dots & \end{aligned}$$

In the source code below, we use five Taylor terms at a time.

### Source for $\sinh(x)$ using coefficient scaling

```
float_precision s(const float_precision& x)
{
    const int group=5;
    size_t precision = x.precision() + 2+(size_t)ceil(log10(x.precision()));
    size_t loopcnt = 2;
    intmax_t k;
    uintmax_t i;
    float_precision r, sinhx, v(x), vsq, terms;
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

```
const float_precision c1(1), c3(3), c4(4);

if (x.sign() < 0)
    v.change_sign();

// Automatically calculate optimal reduction factor as a power of two
k = 8 * (intmax_t)ceil(log(2)*log(precision));
k += v.exponent() + 2;

// Now use the trisection identity sinh(3x)=sinh(x)(3+4Sinh^2(x))
// until the argument has been reduced 2/3*k times.
// Converting power of 2 to power of 3.
k = 2*(intmax_t)ceil(2.0*k / 3);

// Adjust the precision
precision += k/4;
v.precision(precision);
r.precision(precision);
sinhx.precision(precision);
vsq.precision(precision);
terms.precision(precision);
r = c3;
r = pow(r, float_precision(k)); // Since r and k is an integer this is fast
v /= r;
vsq = v.square();
r = v;
sinhx = v;

if (group == 1)
    { // No Coefficients rescaling
    // Now iterate using Taylor expansion
    for (i = 3;; i += 2, ++loopcnt)
        {
        r *= vsq / float_precision(i*(i - 1));
        if (sinhx + r == sinhx)
            break;
        sinhx += r;
        }
    }
else
    {
    std::vector<float_precision> vn(group); // vn[0] is not used
    std::vector<float_precision> cn(group);

    for (i = 0; i < group; ++i)
        {
        cn[i].precision(precision); vn[i].precision(precision);
        if (i == 1) vn[i] = vsq;
        if (i > 1) vn[i] = vn[i-1]*vsq;
        }
    // Now iterate
    for (i = 3; ; )
        {
        int j;
        for (j = group - 1; j >= 0; --j)
            {
            if (j == group - 1)
                {
```

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

```
        cn[j] = float_precision((i + 2*j-1)*(i+2*j));
    }
    else
    {
        cn[j] = cn[j + 1] * float_precision((i + 2*j-
1)*(i+2*j));
    }
    for (j = 2, terms = cn[1]; j < group; ++j)
        terms += cn[j] * vn[j - 1];
    terms += vn[group - 1];
    r *= vsq / cn[0];
    terms *= r;
    i += 2*group;           // Update term count
    loopcnt += group;
    if (sinhx + terms == sinhx) // Reach precision
        break;             // yes terminate loop
    sinhx += terms;         // Add the Taylor terms to result
    if (group > 1)
        r *= vn[group - 1]; // ajust r to last Taylor term
    }
}

for (; k > 0; k--)
{
    sinhx *= (c3 + c4*sinhx.square());
}

// Round to same precision as argument and rounding mode
sinhx.mode(x.mode());
sinhx.precision(x.precision());
if (x.sign() < 0)
    sinhx.change_sign();
return sinhx;
}
```

## Full coefficient scaling

Now, why stop with the grouping of only a few Taylor terms? In [8], they devised a new computation of the Taylor series (albeit for the  $e^x$  Taylor series), postponing the division until all Taylor terms had been calculated.

We can use the same rationale in [8] for the  $\sinh(x)$ . By noticing that by evaluating the Taylor series backward, you can set this recursion:

$$n!\sinh(x) = x^{n!} + \dots + ((n(n-1)(n-2)(n-2) + (n(n-1) + x^2)x^2)x^2 \dots) \quad (9)$$

Here, you summed up the series and postponed the division to one final division to calculate  $\sinh(x)$ . This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Luckily, figuring out the needed number of Taylor terms is not difficult. We are using Sterling approximation for the factorial. We can write the error terms required for a given decimal precision  $P$ .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (10)$$

Where  $P$  is the needed decimal precision.

We then have our  $f(y)$ :

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (11)$$

Where  $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$ . It is not essential for high accuracy here.

And  $f'(y)$ :

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (12)$$

Applying the Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} => \quad (13)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (14)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (15)$$

Since we need an integral number of Taylor terms that need to be odd, we don't need to carry that much precision. As a starting point, [8] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (16)$$

## Fast Hyperbolic functions for Arbitrary Precision numbers

The starting formula works well for the Newton iteration since  $0 < x < 1$  due to argument reduction. Since we are only interested in the ceiling  $n_{i+1}$ , we will only need a few iterations to find the number of Taylor terms required.

We can, therefore, replace the previous code for coefficient scaling for five Taylor terms with this one.

Source code for  $\sinh(x)$  using full coefficient scaling.

```
// Sinh(x) uses Taylor series, argument reduction, and full coefficient scaling.
//
float_precision sinhM(const float_precision& x)
{
    //const int group = 5;
    size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
    intmax_t k;
    uintmax_t i;
    float_precision sinhx, v(x), vsq;
    const float_precision c1(1), c3(3), c4(4);

    if (x.sign() < 0)
        v.change_sign();    // sinh(-x) == -sinh(x)

    // Automatically calculate optimal reduction factor as a power of two
    k = 8 * (intmax_t)ceil(log(2) * log(precision));
    k += v.exponent() + 2;
    k = std::max((intmax_t)0, k);
    // Now use the trisection identity sinh(3x) = sinh(x)(3 + 4sinh^2(x))
    // until the argument has been reduced 2/3*k times. Converting power of 2
    // to power of 3.
    k = (intmax_t)ceil(2.0 * k / 3);

    // Adjust the precision
    precision += k / 4;
    v.precision(precision);
    sinhx.precision(precision);
    vsq.precision(precision);
    sinhx = c3;
    sinhx = pow(sinhx, float_precision(k)); // Since r and k is an integer,
    // this will be faster
    v /= sinhx;
    vsq = v.square();
    sinhx = v;

    // How many Taylor terms? Use Newton's method to find out
    // M is the Taylor term of x^M/M! which is no the first M terms but only
    // the first M/2 terms
    // M need to be odd to comply with the Taylor series for Sinh (x)
    double M;
    const double log_abs_x = log(abs(double(v)));
    M = precision * log(10) / (log(precision * log(10) - log_abs_x - log(abs(-
    log_abs_x)))));
    for (i = 1; ++i)
    {
        const double y = (M + 0.5) * log(M) - M * (log_abs_x + 1.0) -
        precision * log(10) + 0.22579;
        const double dy = (M + 0.5) / M + log(M) - log_abs_x - 1;
        if (int(M) == int(M - y / dy))
            break;
        M += -y / dy;
    }
}
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

```
int m = int(M + 1);
m += m & 0x1 ? 0 : 1; // Ensure the odd terms

// Do (m+1)/2 Taylor terms in reverse order and defer the division to after
the Taylor terms looping
loopcnt_sinh = (m + 1) / 2;
int_precision mFak(1);
sinhx = vsq;
for (i = m; i >=3; i-=2)
{
    mFak *= int_precision(i*(i-1)); // Build the next m*(m-1)
    sinhx += float_precision(mFak, precision);
    if (i == 3) // in last loop
        sinhx *= v;
    else
        sinhx *= vsq;
}
sinhx /= float_precision(mFak, precision);

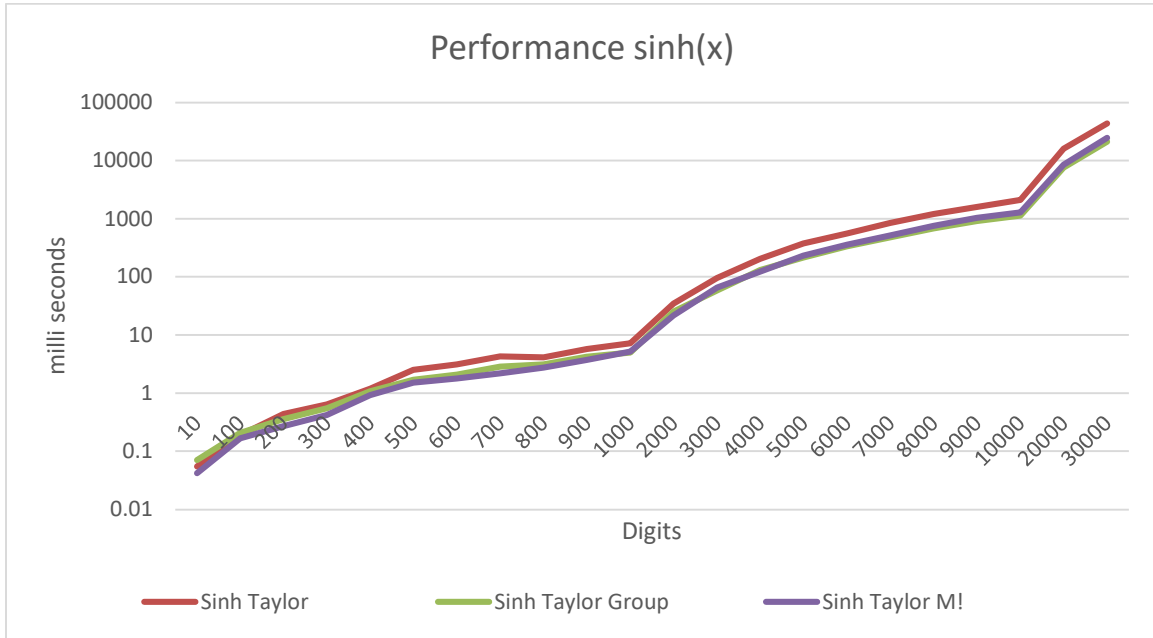
// Reverse argument reduction
for (; k > 0; k--)
    sinhx *= (c3 + c4 * sinhx.square());

// Round to same precision as argument and rounding mode
sinhx.mode(x.mode());
sinhx.precision(x.precision());
if (x.sign() < 0)
    sinhx.change_sign();
return sinhx;
}
```

### *sinh(x) Performance*

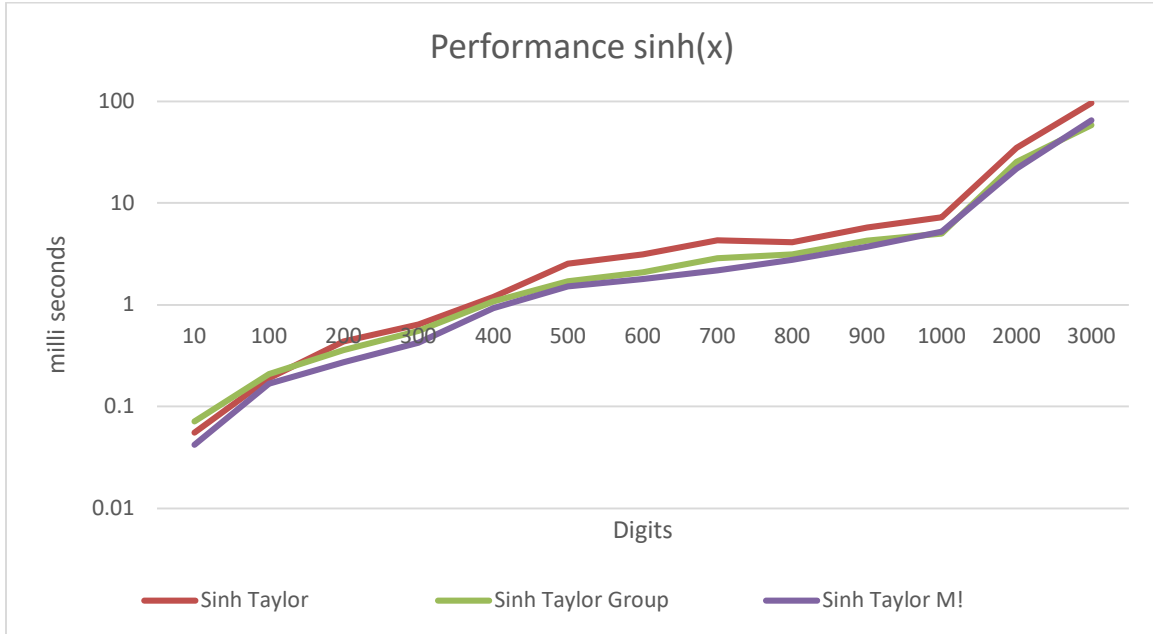
Below is the performance of the three mentioned  $\sinh(x)$  computation methods. The performance chart shows the performance in milliseconds for a precision ranging from 10 decimals to 30,00 digits. The y-axis is a logarithm scale, and computing one Taylor term at a time is considerably slower than the other methods ( $\sinh$  Taylor). Roughly half the speed of the Taylor series, grouping five terms at a time, and the Taylor series, where we postponed the division until after the computation of the Taylor terms. ( $\sinh$  Taylor M!)

# Fast Hyperbolic functions for Arbitrary Precision numbers



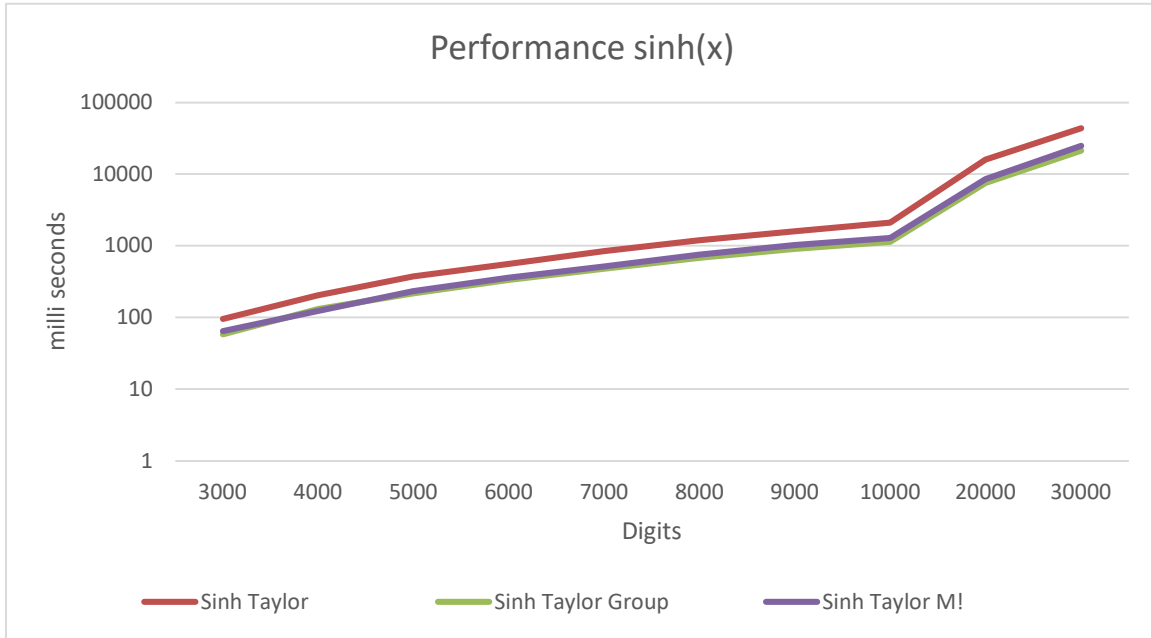
sinh(x) performance between 10 to 30,000 decimal digits computation.

The performance between partial coefficient scaling (Taylor Group) and full coefficient scaling (Taylor M!) is much closer. Taylor M! is fastest up to approximately 1,000-2,000 decimal digits, whereas Taylor Group takes over. However, only around 10% faster.



sinh(x) performance between 10 to 3,000 decimal digits computation.

# Fast Hyperbolic functions for Arbitrary Precision numbers



sinh(x) performance between 3,000 to 30'000 decimal digits computation.

## ***Recommendation for calculating sinh (x)***

Based on the performance measure of the various sinh (x) methods recommended:

- Use standard Taylor series for sinh (x) with an aggressive reduction factor to speed up the Taylor terms calculation.
- Use coefficient scaling to increase performance.
  - a. Either use the full coefficient scaling or
  - b. The partial coefficient scaling.
- The sinh(x) full coefficient scaling is the fastest to around 1,000-2,000 decimal digits.
- The sinh(x) partial coefficient scaling is fastest above 1,000-2,000 decimal digits.

## **Cosh(x) using Exp(x)**

It is tempting to use the definition of cosh(x):

$$\cosh(x) = \frac{1}{2}(e^x + e^{-x}) = \frac{1}{2}\left(e^x + \frac{1}{e^x}\right) \quad (17)$$

We only need to calculate  $e^x$  once. If you have a fast implementation of  $e^x$ , you can use the above to calculate  $\cosh(x)$  and save some code. However, the Taylor series for  $\cosh(x)$  is faster to compute than using the Taylor series for  $\exp(x)$ .

# Fast Hyperbolic functions for Arbitrary Precision numbers

Example of Cosh(x) using exp(x)

```
float_precision coshExp(const float_precision& x)
{
    float_precision v;
    const float_precision c1(1);

    v.precision(x.precision());
    v = exp(x);
    v += c1 / v;
    v.adjustExponent(-1);    // v *= 0.5;

    // Round to same precision as argument and rounding mode
    v.mode(x.mode());
    v.precision(x.precision());
    return v;
}
```

## Cosh(x) using Taylor series:

For cosh(x), we again use a Taylor series until any additional addition does not change the result for the given precision of the number.

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{for any real value } x \quad 18)$$

Example cosh(1):

We are calculating cosh(1) using no argument reduction. We need nine Taylor terms to get the result using the Taylor series.

Cosh(x)		Original	X Reduced	
x=		1	1	
Taylor reductions=		0		
Terms	Term value	Taylor sum	Cosh(x)	Error
1	1.00E+00	1	1.000000000000000	5.43E-01
2	5.00E-01	1.5	1.500000000000000	4.31E-02
3	4.17E-02	1.54166667	1.541666666666667	1.41E-03
4	1.39E-03	1.54305556	1.543055555555556	2.51E-05
5	2.48E-05	1.54308036	1.54308035714286	2.78E-07
6	2.76E-07	1.54308063	1.54308063271605	2.10E-09
7	2.09E-09	1.54308063	1.54308063480373	1.15E-11
8	1.15E-11	1.54308063	1.54308063481520	4.77E-14
9	4.78E-14	1.54308063	1.54308063481524	0.00E+00

Most issues arising in arbitrary precision for sinh (x) also apply to cosh(x).

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## Argument Reduction

Looking at the Taylor series for  $\cosh(x)$ , it is clear that we prefer to have our  $|x| < 1$  to ensure that the Taylor series converges more quickly. As we have seen before, we can use *argument reduction* to work with a smaller number to get a faster converging of  $\cosh(x)$  using fewer *Taylor terms* of the Taylor series.

We can use the trisection identity:

$$\cosh(3x) = \cosh(x)(4\cosh^2(x) - 3) \quad (19)$$

To reduce the argument with a factor of three, after the Taylor iterations, we restore and find the correct value for  $\cosh(x)$  by applying this formula the same number of times we did when reducing the argument.

### Example – Two-argument reduction:

Using the same example as before for  $\cosh(1)$  and using two argument reductions, you get the result after only five Taylor terms compared to nine with no argument reductions.

Cosh(x)		Original	X Reduced		
x=		1	0.111111111		
Taylor reductions=		2			
Terms	Term value	Taylor sum	Cosh(x)	Error	
1	1.00E+00	1	1.000000000000000	5.43E-01	
2	6.17E-03	1.00617284	1.54247714909996	6.03E-04	
3	6.35E-06	1.00617919	1.54308038649498	2.48E-07	
4	2.61E-09	1.00617919	1.54308063476051	5.47E-11	
5	5.76E-13	1.00617919	1.54308063481524	2.00E-15	

### Example – Eight-argument reductions:

With eight times argument reduction, you get the result after two Taylor terms compared to five using two argument reductions. However, the error is considerably higher (less accurate) than the equivalent calculation for  $\sinh(1)$ .

Cosh(x)		Original	X Reduced		
x=		1	0.000152416		
Taylor reductions=		8			
Terms	Term value	Taylor sum	Cosh(x)	Error	
1	1.00E+00	1	1.000000000000000	5.43E-01	
2	1.16E-08	1.00000001	1.54308063305537	1.76E-09	

Unsurprisingly, using argument reduction significantly reduced the number of Taylor terms needed and will result in faster performance. However, aggressive reductions in argument result in a significant decrease in accuracy. This is due to the trisection identity, and the Taylor sum is approaching one for a tiny argument, resulting in a higher loss of

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

accuracy unless you take precautions. To avoid inaccuracy in the result, we increase the precision by  $k$  (instead of  $k/4$  as for  $\sinh(x)$ ).

### *Cosh(x) using double angle reduction*

Argument reduction reduces  $x$  to a much smaller value that is more sensitive to round-off errors for  $\cosh(x)$  than its counterpart for  $\sinh(x)$ . It is, therefore, potentially better to use the double-angle formula:

$$\cosh(2x) = \cosh^2(x) - 1 \quad (20)$$

Alternatively, it is even better written as:

$$\cosh(2x) = 2(1 - \cosh(x))^2 - 4(1 - \cosh(x)) + 1 \quad (21)$$

Although it does not prevent round-off errors, it is less sensitive than the trisection formula. We calculate the reduction factor for  $\cosh(x)$  as  $k = 8[\ln(2) * \ln(P)]$  for higher precisions, and then we adjust for the magnitude of  $x$ . We add the exponent to the reduction factor. This means that our reduction factor gets smaller if  $x$  is very small, preventing us from making unnecessary reductions. If  $x$  is very small, the reduction factor is negative, and we do not perform any argument reductions.

Since we are slightly less sensitive using the double angle formula versus the trisection formula, we only increase the precision by  $0.75 * k$ .

### *Source for cosh(x) with argument reduction and coefficient scaling*

```
float_precision cos(const float_precision& x)
{
    const int group = 5;
    size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
    intmax_t k, i;
    uintmax_t loopcnt = 1;
    float_precision r, cosx, v(x), vsq, terms;
    const float_precision c1(1), c2(2), c4(4);

    // Check for argument reduction and increase precision if necessary
    // Automatically calculate optimal reduction factor as a power of two
    k = 2 * (intmax_t)ceil(log(2)*log(precision));

    precision += 3*k/4;
    r.precision(precision);
    cosx.precision(precision);
    v.precision(precision);
    vsq.precision(precision);
    terms.precision(precision);

    // Check that the argument is larger than 2*PI and reduce it if needed.
    // No need for high precision.
    // we just need to figure out if we need to Calculate PI with
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

```
// a higher precision
if (abs(v) > float_precision(2 * 3.14159265))
    { // Reduce argument to between 0..2P
    cosx = _float_table(_PI, precision);
    cosx.adjustExponent(-1); // Multiply with 2
    if (abs(v) > cosx)
        {
        r = v / cosx;
        (void)modf(r, &r);
        v -= r * cosx;
        }
    if (v < float_precision(0))
        v += cosx;
    }

// Reduced it further to between 0..PI.
// However, avoid calculating PI is not needed.
// No need for high precision.
//We just need to figure out if we need to Calculate PI with
// a higher precision
if (abs(v) > float_precision(3.14159265))
    {
    r = _float_table(_PI, precision);
    if (v > r)
        v = r * c2 - v; // cos(x)=cos(2PI - x) for x >= PI
    }

// Adjust k for the final value of v when v is small (less than 1).
// We know it is in the interval between [0..PI]
// This indicates that the exponent is in the range [-inf..1]
// Avoid unnecessary argument reduction if v is small
k += v.exponent();
k = std::max((intmax_t)0, k);

// Now use the double identity cos(2x)=2cos(x)^2-1
// k times k. Where k is the number of reduction factor-based
// on the needed precision of the argument.
v.adjustExponent(-k); // Divide with 2^k
vsq = v.square();
r = c1;
cosx = r;

if (group == 1)
    {
    // Now iterate using Taylor expansion
    for (i = 2;; i += 2, ++loopcnt)
        {
        r *= vsq / float_precision(i*(i - 1));
        r.change_sign();
        if (cosx + r == cosx)
            break;
        cosx += r;
        }
    }
else
    {
    std::vector<float_precision> vn(group); // vn[0] is not used
    std::vector<float_precision> cn(group); //
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

```
for (i = 0; i < group; ++i)
{
    cn[i].precision(precision); vn[i].precision(precision);
    if (i == 1) vn[1] = vsq;
    if (i > 1) vn[i] = vn[i - 1] * vsq;
}
// Now iterate
for (i = 2; ; )
{
    // Re-calculate the coefficients
    intmax_t j;
    for (j = group - 1; j >= 0; --j)
    {
        if (j == group - 1)
        {
            cn[j] = float_precision((i + 2 * j - 1)*(i + 2 *
j), precision);

            if ((i / 2 + j - 1) & 0x1) // Odd
                cn[j].change_sign();
        }
        else
        {
            cn[j] = -cn[j + 1] * float_precision((i + 2 * j
- 1)*(i + 2 * j), precision);
        }
    }

    cn[0] = abs(cn[0]).inverse();
    // Adding from smallest to largest number
    terms = vn[group - 1];
    if ((i / 2 + group - 1) & 0x1)
        terms.change_sign();
    for (j = group - 1; j >= 2; --j)
        terms += cn[j] * vn[j - 1];
    terms += cn[1];
    r *= vsq*cn[0];
    terms *= r;
    i += 2 * group; // Update term count
    loopcnt += group;
    if (cosx + terms == cosx) // Reach precision
        break; // yes terminate loop
    cosx += terms; // Add Taylor terms to result
    if (group > 1)
        r *= vn[group - 1]; // ajust r to last Taylor term
}

// Double formula cos(2x)=cos^2(x)-1=-4(1-cos(x)+2*(1-cosx)^2+1
for (; k > 0; --k)
{
    v = c1 - cosx;
    cosx = -c4*v + c2*v.square() + c1;
}

// Round to same precision as argument and rounding mode
cosx.mode(x.mode());
cosx.precision(x.precision());
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

```
return cosx;  
}
```

The performance is similar to the  $\cosh(x)$  using the trisection as a reduction factor. Although you can use a little bit less precision, it does not change the performance observed compared to the trisection formula of  $\cosh(x)$ .

### Coshx(x) with full coefficients scaling

As we saw with  $\sinh(x)$ , we can also apply the full coefficient scaling to  $\cosh(x)$ .  $\cosh(x)$  is very similar to the  $\sinh(x)$ .

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{ for any real value } x \quad (22)$$

By evaluating the Taylor series backward, you can set this recursion:

$$n! \cosh(x) = n! + \dots + ((n(n-1)(n-2)(n-2) + (n(n-1) + x^2)x^2)x^2)x^2 \dots \quad (23)$$

Here, you summed up the series and postponed the division to one final division to calculate  $\cosh(x)$ . This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Luckily, figuring out the needed number of Taylor terms is not difficult. We are using Sterling approximation for the factorial. We can write the error terms required for a given decimal precision  $P$ .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (24)$$

Where  $P$  is the needed decimal precision.

We then have our  $f(y)$ :

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (25)$$

Where  $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$ . We don't need to carry this with high precision.

And  $f'(y)$ :

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (26)$$

Applying the Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (27)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (28)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (29)$$

Since we need an integral number of Taylor terms that need to be even, we don't need to carry that much precision. As a starting point, [9] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (30)$$

The starting formula works well for the Newton iteration since  $0 < x < 1$  due to argument reduction. Since we are only interested in the ceiling  $n_{i+1}$ , we will only need a few iterations to find the number of Taylor terms required.

We can, therefore, replace the previous code for coefficient scaling for five Taylor terms with this one for  $\cosh(x)$  using full coefficient scaling.

Source code for  $\cosh(x)$  using Taylor series with argument reduction and full coefficient scaling.

```
// Cosh(x) using Taylor series, argument reduction, and full coefficient scaling.
//
float_precision testcoshM(const float_precision& x, const int group = 1)
{
    //const int group = 3;
    size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
    size_t loopcnt = 1;
    intmax_t k;
    uintmax_t i;
    float_precision coshx, v(x), vsq, terms;
    const float_precision c1(1), c2(2), c3(3), c4(4);

    if (x.sign() < 0)
        v.change_sign(); // cosh(-x) = cosh(x)

    // Automatically calculate optimal reduction factor as a power of two
    k = 8 * (intmax_t)ceil(log(2) * log(precision));
    k += v.exponent() + 2;
    // Now use the trisection identity cosh(3x)=cosh(x)(4Cosh^2(x)-3)
```

## Fast Hyperbolic functions for Arbitrary Precision numbers

```
// until argument has been reduced 2/3*k times. Converting power of 2 to
power of 3.
k = (intmax_t)ceil(2.0 * k / 3);

// Adjust the precision
precision += k;
coshx.precision(precision);
v.precision(precision);
vsq.precision(precision);
terms.precision(precision);

coshx = c3;
coshx = pow(coshx, float_precision(k)); // Since coshx and k is an integer
this will be fast
v /= coshx; // argument reduction with x/(3^k)
vsq = v.square();

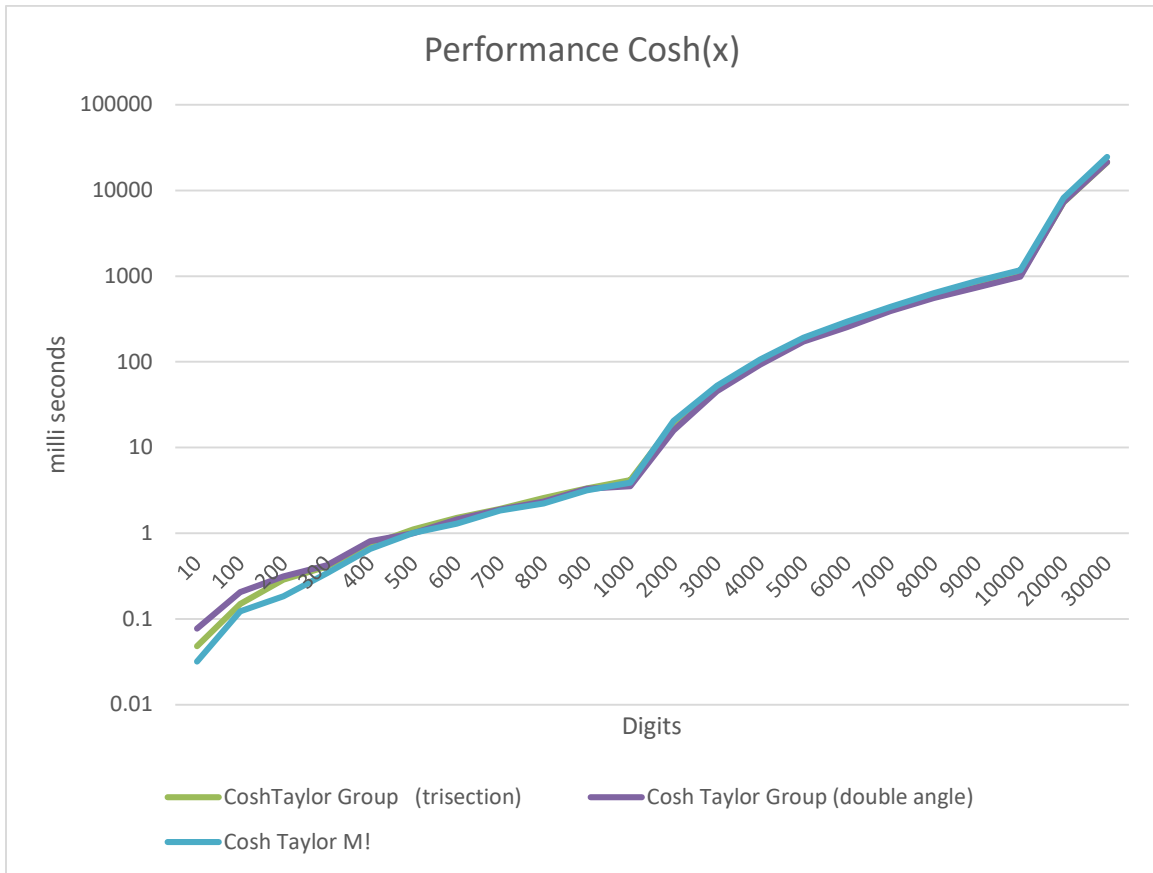
// How many Taylor terms?. Use Newton method to find out
// M is the Taylor terms of x^M/M! which is no the first M terms but only
the first M/2 terms
// M need to be even to comply with the Taylor series for cosh(x)
double M;
const double log_abs_x = log(abs(double(v)));
M = precision * log(10) / (log(precision * log(10) - log_abs_x - log(abs(-
log_abs_x)))));
for (i = 1; ++i)
{
    const double y = (M + 0.5) * log(M) - M * (log_abs_x + 1.0) -
precision * log(10) + 0.22579;
    const double dy = (M + 0.5) / M + log(M) - log_abs_x - 1;
    if (int(M) == int(M - y / dy))
        break;
    M += -y / dy;
}
int m = int(M + 1);
m += m & 0x1 ? 1 : 0; // Ensure the even terms

// Do (m+1)/2 Taylor terms in reverse order and deferred the division to
after the Taylor terms looping
loopcnt_cosh = m / 2;
int_precision mFak(1);
coshx = vsq;
for (i = m; i >= 4; i -= 2)
{
    mFak *= int_precision(i * (i - 1)); // Build the next m*(m-1)
    coshx += float_precision(mFak, precision);
    coshx *= vsq;
}
mFak *= c2;
coshx /= float_precision(mFak, precision);
coshx += c1; // add the first Taylor term last!

// Reverse argument reduction
for (; k > 0; k--)
    coshx *= (c4 * coshx.square() - c3);

// Round to same precision as argument and rounding mode
coshx.mode(x.mode());
coshx.precision(x.precision());
return coshx;
}
```

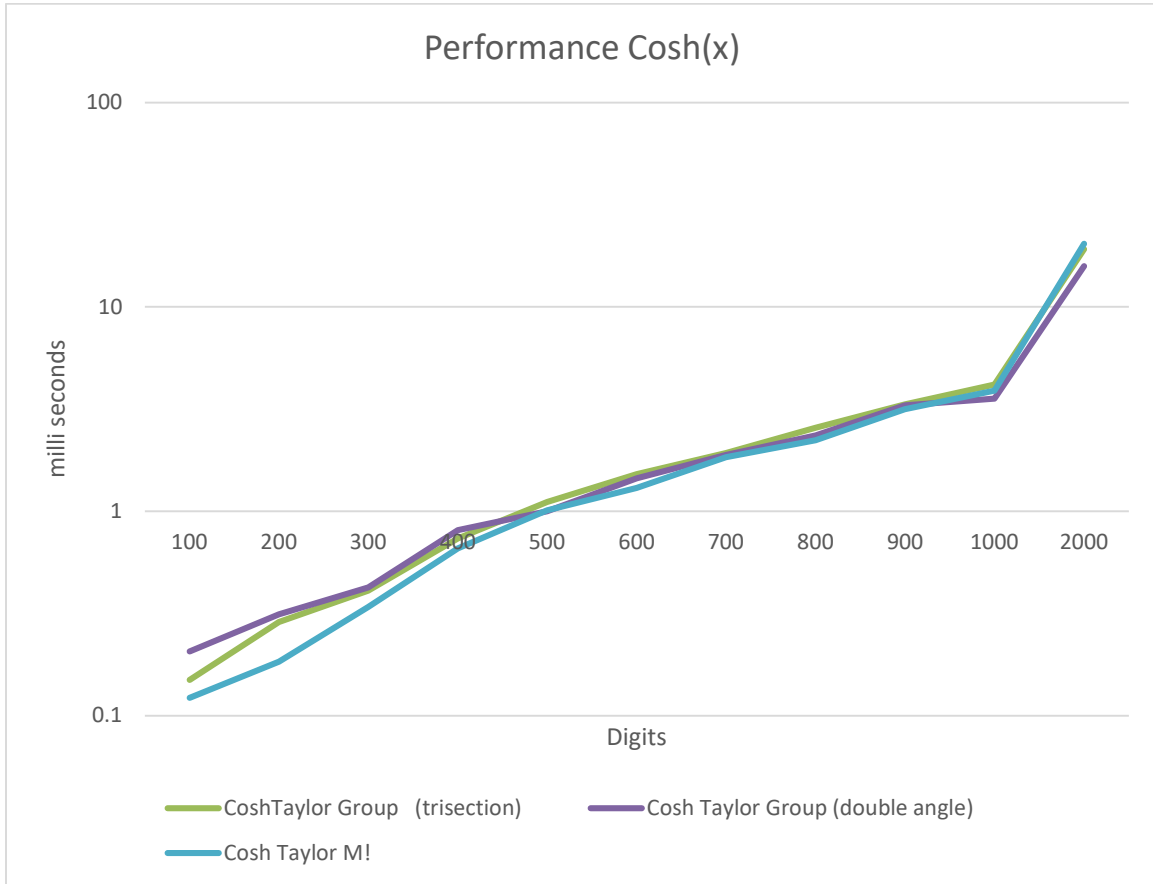
## *Cosh(x) Performance*



Cosh(x) performance for digits in the range of 10 to 30,000.

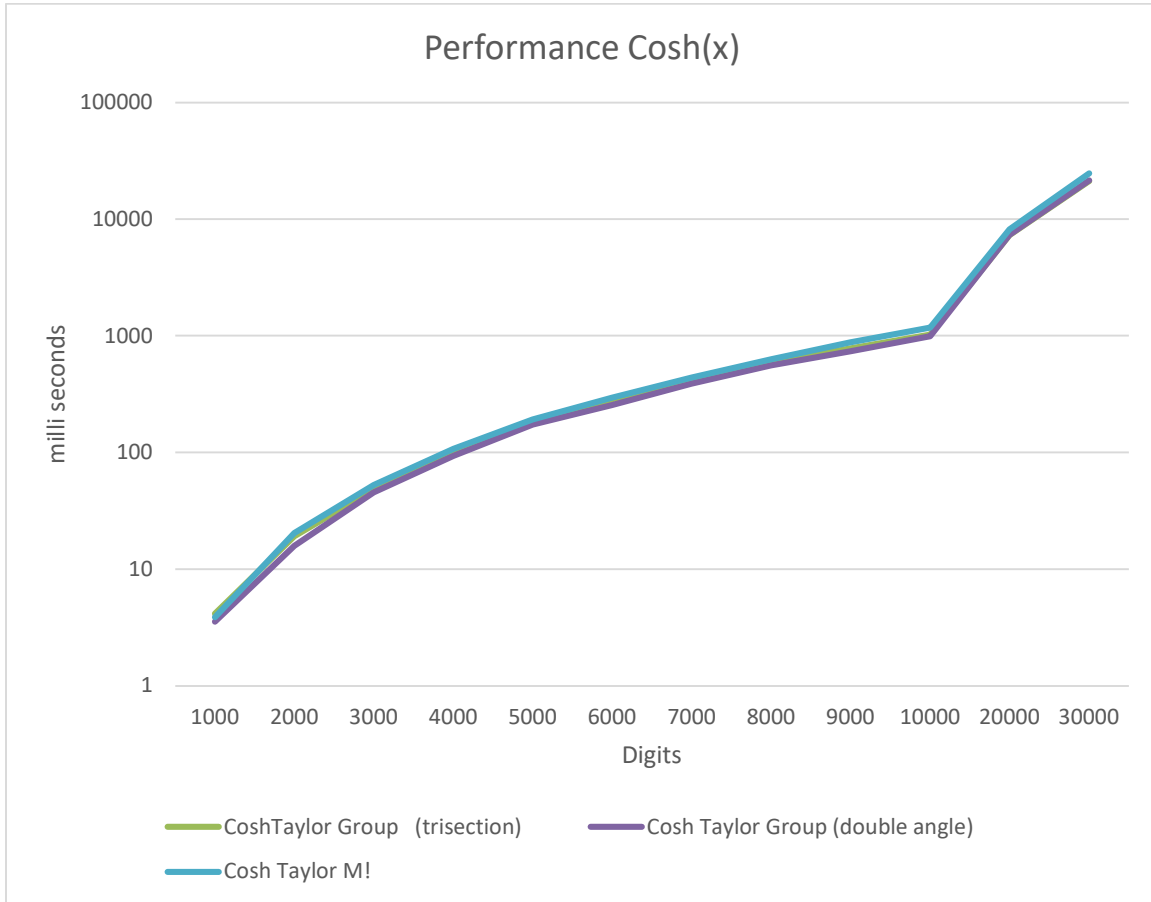
It is hard to see any difference between Cosh Taylor Group and cosh(x) Taylor M!

# Fast Hyperbolic functions for Arbitrary Precision numbers



Cosh(x) in the range of 100 to 2,000 digits.

# Fast Hyperbolic functions for Arbitrary Precision numbers



Cosh(x) in the range from 1,000 to 30,000 digits.

## ***Recommendation for calculating cosh(x)***

Based on the performance measure of the various cosh(x) methods, recommend:

- It is a matter of taste whether to use the cosh(x) using the double angle formula or the trisection formula since the performance is equivalent.
- Do not use the Taylor series for cosh(x) with an aggressive reduction factor to speed up the Taylor term calculation.
- Use coefficient scaling to increase performance—particularly full coefficient scaling below 1,000 digits and partial coefficient scaling above 1,000 digits.

## **Tanh(x):**

Tanh(x) is defined as:

---

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (31)$$

Which seems to be the most effective way of calculating  $\tanh(x)$  by using one call for  $\exp(x)$ .

We could also use the Taylor series for  $\tanh(x)$ :

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \dots \frac{(-1)^{n-1} \cdot 2^{2n} (2^{2n} - 1) \beta_n x^{2n-1}}{(2n)!} + \dots \quad (32)$$

Where  $B_n$  is the Bernoulli number, however, since we do not know how many Bernoulli numbers we need, this will require us to calculate Bernoulli numbers on the fly, which is much more complicated to implement than just a call to the  $e^{2x}$  function.

### Source for $\tanh(x)$

```
float_precision tanh( const float_precision& x )
{
    float_precision v(x), vsq;
    const float_precision c1(1);

    v.precision( x.precision() + 1 );
    vsq.precision( x.precision() + 1 );
    v = exp( v );
    vsq = v.square();
    v = (vsq - c1) / (vsq + c1);

    // Round to the same precision as argument and rounding mode
    v.mode( x.mode() );
    v.precision( x.precision() );
    return v;
}
```

### Recommendation for calculating $\tanh(x)$

Based on the performance measure of the various  $\tanh(x)$  methods, we recommend the following:

- Use the definition of  $\tanh(x)$  using  $\exp(x)$  in favor of using the Taylor series for  $\tanh(x)$ .

### Arcsinh(x):

There are two methods: the direct method and the Taylor series method.

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## ***Arcsinh(x) direct method:***

Arcsinh(x) is equal to:

$$\operatorname{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1}) \quad (33)$$

Ln(x) is relatively fast to calculate; the same goes for the square root. This direct method is the preferred way of calculating Arcsinh(x).

Source for asinh(x)

```
float_precision asinh( const float_precision& x )
{
    float_precision v(x);
    const float_precision c1(1);

    v.precision( x.precision() + 1 );
    v = log(v+sqrt(v.square()+c1));

    // Round to the same precision as argument and rounding mode
    v.mode( x.mode() );
    v.precision( x.precision() );
    return v;
}
```

## ***Arcsinh(x) using the Taylor series***

Arcsinh(x) also has a Taylor series equivalence based on two Taylor series. This first Taylor series is for  $|x| < 1$ :

$$\operatorname{Arcsinh}(x) = x - \frac{1}{2} \frac{x^3}{3} + \frac{1 \cdot 3}{2 \cdot 4} \frac{x^5}{5} - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{x^7}{7} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{x^9}{9} - \dots \quad (34)$$

The second Taylor series is for  $|x| \geq 1$ :

$$\operatorname{Arcsinh}(x) = \pm \ln(2x) + \frac{1}{2} \frac{1}{2x^2} - \frac{1 \cdot 3}{2 \cdot 4} \frac{1}{4x^4} + \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{1}{6x^6} - \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{1}{8x^8} - \dots \quad (35)$$

The + applies for  $x \geq 1$  and – for  $x \leq -1$ .

Example: Arcsinh(0.1)

A result is found using only seven Taylor terms

ArcSinh(x)		Original		$ x  < 1$
Terms	x=	0.1	ArcSinh(x)	Error
1	1.00E-01	0.10000000000000	0.1000000000000000	-1.66E-04
2	1.67E-04	0.09983333333333	0.0998333333333333	7.46E-07
3	7.50E-07	0.09983408333333	0.0998340833333333	-4.43E-09

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

4	4.46E-09	0.0998340788690	0.099834078869048	3.02E-11
5	3.04E-11	0.0998340788994	0.099834078899430	-2.22E-13
6	2.24E-13	0.0998340788992	0.099834078899206	1.73E-15
7	1.74E-15	0.0998340788992	0.099834078899208	0.00E+00

### Example: Arcsinh(0.7)

A result is found after 27 Taylor Terms, but the result is not very accurate (error  $\sim 10^{-12}$ )

ArcSinh(x)	x=	Original	x <1	
Terms	Term value	Term Sum	ArcSinh(x)	Error
		<b>0.7</b>		
1	7.00E-01	0.700000000000000	0.700000000000000	-4.73E-02
2	5.72E-02	0.642833333333333	0.642833333333333	9.83E-03
3	1.26E-02	0.655438583333333	0.655438583333333	-2.77E-03
4	3.68E-03	0.651762052083333	0.651762052083333	9.05E-04
5	1.23E-03	0.6529880731293	0.652988073129340	-3.22E-04
6	4.42E-04	0.6525457024446	0.652545702444649	1.21E-04
7	1.68E-04	0.6527138316619	0.652713831661927	-4.73E-05
8	6.63E-05	0.6526475327072	0.652647532707247	1.90E-05
9	2.69E-05	0.6526744057210	0.652674405721047	-7.84E-06
10	1.11E-05	0.6526632785647	0.652663278564660	3.29E-06
11	4.69E-06	0.6526679649520	0.652667964952025	-1.40E-06
12	2.00E-06	0.6526659636053	0.652665963605294	6.02E-07
13	8.65E-07	0.6526668282204	0.652666828220438	-2.62E-07
14	3.77E-07	0.6526664510290	0.652666451029002	1.15E-07
15	1.66E-07	0.6526666169607	0.652666616960717	-5.09E-08
16	7.35E-08	0.6526665434351	0.652666543435125	2.26E-08
17	3.28E-08	0.6526665762216	0.652666576221552	-1.01E-08
18	1.47E-08	0.6526665615197	0.652666561519732	4.56E-09
19	6.63E-09	0.6526665681449	0.652666568144933	-2.06E-09
20	3.00E-09	0.6526665651461	0.652666565146113	9.36E-10
21	1.36E-09	0.6526665665089	0.652666566508912	-4.27E-10
22	6.22E-10	0.6526665658874	0.652666565887360	1.95E-10
23	2.84E-10	0.6526665661718	0.652666566171770	-8.94E-11
24	1.31E-10	0.6526665660412	0.652666566041240	4.11E-11
25	6.01E-11	0.6526665661013	0.652666566101311	-1.90E-11
26	2.77E-11	0.6526665660736	0.652666566073596	8.76E-12
27	1.28E-11	0.6526665660864	0.652666566086412	-4.06E-12

There are several issues with the Taylor series approach.

- First, the Taylor series convergence slowly when x is approaching one from either side.
- Secondly, you need to calculate the  $\ln(2x)$ , which takes approximately the same time as the direct method.

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

- Thirdly, you cannot overcome slow convergence since no argument reduction is available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series method is not recommended for  $\text{Arcsinh}(x)$ .

### ***Recommendation for calculating $\text{Arcsinh}(x)$***

Based on the performance measure of the various  $\text{arsinh}(x)$  methods, we recommend the following:

- Use the Direct method:  $\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$

### **$\text{Arccosh}(x)$ :**

Again, there are two methods: the direct method and the method using the Taylor series.

#### ***$\text{Arccosh}(x)$ direct method:***

$\text{Arccosh}(x)$  is equal to:

$$\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1}), \text{ where } x \geq 1 \quad (36)$$

$\ln(x)$  is relatively fast to calculate, and the same goes for the square root. This direct method is the preferred way of calculating  $\text{Arccosh}(x)$ .

Source for  $\text{Arccosh}(x)$

```
float_precision acosh( const float_precision& x )
{
    float_precision v(x);
    const float_precision c1(1);

    if( x < c1 )
        { throw float_precision::domain_error(); }

    v.precision( x.precision() + 1 );
    v = log(v+sqrt(v.square()-c1));

    // Round to the same precision as argument and rounding mode
    v.mode( x.mode() );
    v.precision( x.precision() );
    return v;
}
```

## *Arccosh(x) using the Taylor series*

Arccosh(x) also have a Taylor series equivalence for  $|x| \geq 1$ :

$$\text{Arccosh}(x) = \ln(2x) - \left( \frac{1}{2} \frac{1}{2x^2} + \frac{1 \cdot 3}{2 \cdot 4} \frac{1}{4x^4} + \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \frac{1}{6x^6} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{2 \cdot 4 \cdot 6 \cdot 8} \frac{1}{8x^8} - \dots \right) \quad (37)$$

However, it converges just as slowly as the Arcsinh(x).

### Example: Arccosh(5)

A result is found using nine Taylor terms.

ArcCosh(x)		Original	x>=1	
Terms	x=	5	ArcCosh(x)	Error
1	2.30E+00	2.30258509299405	2.30258509299405	-1.02E-02
2	1.00E-02	2.29258509299405	2.29258509299405	-1.53E-04
3	1.50E-04	2.29243509299405	2.29243509299405	-3.42E-06
4	3.33E-06	2.29243175966071	2.29243175966071	-9.01E-08
5	8.75E-08	2.29243167216071	2.29243167216071	-2.60E-09
6	2.52E-09	2.29243166964071	2.29243166964071	-7.95E-11
7	7.70E-11	2.29243166956371	2.29243166956371	-2.53E-12
8	2.45E-12	2.29243166956126	2.29243166956126	-8.30E-14
9	8.04E-14	2.29243166956118	2.29243166956118	0.00E+00

### Example: Arccosh(2)

A result is found using twenty-one Taylor terms

ArcCosh(x)		Original	x>=1	
Terms	x=	2	ArcCosh(x)	Error
1	1.39E+00	1.38629436111989	1.38629436111989	-6.93E-02
2	6.25E-02	1.32379436111989	1.32379436111989	-6.84E-03
3	5.86E-03	1.31793498611989	1.31793498611989	-9.77E-04
4	8.14E-04	1.31712118403656	1.31712118403656	-1.63E-04
5	1.34E-04	1.31698766963226	1.31698766963226	-2.98E-05
6	2.40E-05	1.31696363703949	1.31696363703949	-5.74E-06
7	4.59E-06	1.31695904748184	1.31695904748184	-1.15E-06
8	9.13E-07	1.31695813425353	1.31695813425353	-2.37E-07
9	1.87E-07	1.31695794697038	1.31695794697038	-5.00E-08
10	3.93E-08	1.31695790766404	1.31695790766404	-1.07E-08
11	8.40E-09	1.31695789926231	1.31695789926231	-2.34E-09
12	1.82E-09	1.31695789743962	1.31695789743962	-5.15E-10
13	4.00E-10	1.31695789703933	1.31695789703933	-1.15E-10
14	8.88E-11	1.31695789695050	1.31695789695050	-2.57E-11

## Fast Hyperbolic functions for Arbitrary Precision numbers

---

15	1.99E-11	1.31695789693062	1.31695789693062	-5.81E-12
16	4.48E-12	1.31695789692614	1.31695789692614	-1.32E-12
17	1.02E-12	1.31695789692512	1.31695789692512	-3.02E-13
18	2.33E-13	1.31695789692489	1.31695789692489	-6.95E-14
19	5.34E-14	1.31695789692483	1.31695789692483	-1.62E-14
20	1.23E-14	1.31695789692482	1.31695789692482	-4.00E-15
21	2.85E-15	1.31695789692482	1.31695789692482	0.00E+00

Arccosh(x) suffers from the same deficit as the Taylor series for Arcsinh(x).

- First, the Taylor series convergence slowly when x is approaching one.
- Secondly, you need to calculate the  $\ln(2x)$ , which takes approximately the same time as the direct method.
- Thirdly, you cannot overcome slow convergence since no argument reduction is available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series method for Arccosh(x) is not recommended.

### ***Recommendation for calculating Arccosh(x)***

Based on the performance measure of the various arcsinh(x) methods, we recommend the following:

- Use the Direct method:  $\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$

### **Arctanh(x):**

There are two different methods to use. One is the standard Taylor series, and the other is the direct method.

#### ***Arctanh(x) direct method***

Arctanh(x) is equal to:

$$\text{Arctanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right), \quad \text{where } |x| < 1 \quad (38)$$

$\ln(x)$  is relatively fast. This direct method is the preferred way of calculating Arctanh(x).

# Fast Hyperbolic functions for Arbitrary Precision numbers

## Source for Arctanh(x)

```
float_precision atanh( const float_precision& x )
{
    float_precision v(x);
    const float_precision c1(1);

    if( x >= c1 || x <= -c1 )
        { throw float_precision::domain_error(); }

    v.precision( x.precision() + 1 );
    v = log((c1+v)/(c1-v));
    v.adjustExponent(-1); // v *= c05;

    // Round to the same precision as argument and rounding mode
    v.mode( x.mode() );
    v.precision( x.precision() );
    return v;
}
```

## Arctanh(x) using the Taylor series

For arctanh(x), we can use a Taylor series until any additional addition does not change the result for the given precision of the number:

$$\text{Arctanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} - \dots, \text{ where } |x| < 1 \quad (39)$$

Notice the similarity with the arctan(x) Taylor series. Arctanh(x) uses no alternating signs between Taylor terms as the arctan(x) Taylor series does.

Arctanh(x) suffers from the same weakness as the other hyperbolic function, that there is no argument reduction formula to lower the argument, x, and increase the performance of the Taylor series.

The Arctanh(x) Taylor series converges slowly, mainly when x is close to 1 or -1.

### Example Arctanh(0.1)

We need only seven Taylor terms to get the result.

ArcTanh(x)		Original		x <1	
	x=	0.1			
Terms	Term value	Term Sum	ArcCosh(x)	Error	
1	1.00E-01	0.1000000000000000	0.1000000000000000	3.35E-04	
2	3.33E-04	0.1003333333333333	0.1003333333333333	2.01E-06	
3	2.00E-06	0.1003353333333333	0.1003353333333333	1.44E-08	
4	1.43E-08	0.100335347619048	0.100335347619048	1.12E-10	
5	1.11E-10	0.100335347730159	0.100335347730159	9.17E-13	
6	9.09E-13	0.100335347731068	0.100335347731068	7.79E-15	
7	7.69E-15	0.100335347731076	0.100335347731076	0.00E+00	

# Fast Hyperbolic functions for Arbitrary Precision numbers

---

## Example Arctanh(0.5)

Now we need 23 Taylor terms to get the result. More than three times as many as for arctanh(0.1).

ArcTanh(x)		Original	x <1	
	x=	0.5		
Terms	Term value	Term Sum	ArcTanh(x)	Error
1	5.00E-01	0.5000000000000000	0.5000000000000000	4.93E-02
2	4.17E-02	0.5416666666666667	0.5416666666666667	7.64E-03
3	6.25E-03	0.5479166666666667	0.5479166666666667	1.39E-03
4	1.12E-03	0.549032738095238	0.549032738095238	2.73E-04
5	2.17E-04	0.549249751984127	0.549249751984127	5.64E-05
6	4.44E-05	0.549294141188672	0.549294141188672	1.20E-05
7	9.39E-06	0.549303531212711	0.549303531212711	2.61E-06
8	2.03E-06	0.549305565717919	0.549305565717919	5.79E-07
9	4.49E-07	0.549306014505833	0.549306014505833	1.30E-07
10	1.00E-07	0.549306114892603	0.549306114892603	2.94E-08
11	2.27E-08	0.549306137599134	0.549306137599134	6.73E-09
12	5.18E-09	0.549306142782147	0.549306142782147	1.55E-09
13	1.19E-09	0.549306143974239	0.549306143974239	3.60E-10
14	2.76E-10	0.549306144250187	0.549306144250187	8.39E-11
15	6.42E-11	0.549306144314416	0.549306144314416	1.96E-11
16	1.50E-11	0.549306144329437	0.549306144329437	4.62E-12
17	3.53E-12	0.549306144332965	0.549306144332965	1.09E-12
18	8.32E-13	0.549306144333797	0.549306144333797	2.58E-13
19	1.97E-13	0.549306144333993	0.549306144333993	6.16E-14
20	4.66E-14	0.549306144334040	0.549306144334040	1.50E-14
21	1.11E-14	0.549306144334051	0.549306144334051	3.89E-15
22	2.64E-15	0.549306144334054	0.549306144334054	1.22E-15
23	6.32E-16	0.549306144334054	0.549306144334054	0.00E+00

You can add coefficient scaling to speed things up. However, the Taylor series method is still much slower than the direct method.

It suffers from the same deficit as the Taylor series for Arcsinh(x) and Arccosh(x) but not as bad from a performance perspective.

- First, the Taylor series converges slowly when x is approaching one.
- Secondly, you cannot overcome slow convergence since no argument reduction is available (the technique we use to speed up the Taylor series).
- Lastly, it is several magnitudes slower than the direct method.

Therefore, the Taylor series method for arctanh(x) is not recommended.

## ***Recommendation for calculating Arctanh(x)***

Based on the performance measure of the various arctan(x) methods, we recommend the following:

- Use the direct method.  $\text{Arctanh}(x) = \frac{1}{2} \ln \left( \frac{1+x}{1-x} \right)$ , where  $|x| < 1$

## **Overall Recommendation for calculating Hyperbolic functions**

- Use the Taylor series for calculating sinh(x) using argument reductions and coefficient scaling
- Use the Taylor series for calculating cosh(x) using argument reductions and coefficient scaling
- Use  $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$  for calculating tanh(x)
- Use  $\text{Arcsinh}(x) = \ln(x + \sqrt{x^2 + 1})$  for calculating arcsinh(x)
- Use  $\text{Arccosh}(x) = \ln(x + \sqrt{x^2 - 1})$  for calculating arccosh(x)
- Use  $\text{Arctanh}(x) = \frac{1}{2} \ln \left( \frac{1+x}{1-x} \right)$  for calculating arctanh(x)

## Reference

- 1) Arbitrary precision library package. [Arbitrary Precision C++ Packages \(hvks.com\)](#)
- 2) Numerical recipes in C++, 3<sup>rd</sup> edition, Cambridge University Press, New York, NY 2007
- 3) HVE Fast Log() calculation for arbitrary precision; [Fast Log\(\) calculation for arbitrary precision \(hvks.com\)](#)
- 4) HVE Fast Exp() calculation for arbitrary precision; [Fast Exp\(\) calculation for arbitrary precision \(hvks.com\)](#)
- 5) The Yacas book of algorithms, Version 1.3.3, April 1 2013 by the Yacas team
- 6) Richard Brent & Paul Zimmermann, Modern Computer Arithmetic, Version 0.5.9 17 October 2010; <http://maths-people.anu.edu.au/~brent/pd/mca-cup-0.5.9.pdf>
- 7) The Math behind arbitrary precision for integer and floating-point arithmetic. [The Math behind arbitrary precision \(hvks.com\)](#)
- 8) Ronald W Potter. Arbitrary Precision Calculation of selected higher functions. ISBN #:978-1-312-59943-7